Self-Assembly as Graph Grammar as Distributed System

Aaron Sterling*

July 9, 2009

Abstract

In 2004, Klavins *et al.* introduced the use of graph grammars to describe—and to program— systems of self-assembly. It turns out that these graph grammars are a "dual notion" of a graph rewriting characterization of distributed systems that was proposed by Degano and Montanari over twenty years ago. By applying techniques obtained from this observation, we prove a generalized version of Soloveichik and Winfree's theorem on local determinism, and we also present a canonical method to simulate asynchronous constant-size-message-passing models of distributed computing with systems of self-assembly.

1 Introduction

For millions of years, organisms have assembled themselves into complex beings, using simple building blocks that interact according to local rules. The existence of a central brain is a relatively recent phenomenon in the history of life. However, the history of computation theory is different. Recursion theory was founded in an attempt to capture (and automate) what "a mathematician" does. Early computer science papers presented or analyzed sequential algorithms, and practitioners scheduled time on their institution's sole computer. While distributed computing appeared as a discipline in the 1970's, it only recently has begun to take center stage. This change has been driven by the increasing importance of networked computers and communication devices, but the objective of this paper is to formalize a connection between distributed computing and something much older than the internet: the world of self-assembly, and of the very small.

Two main research areas have studied algorithmic self-assembly: nanotechnology and robotics. Nanostructure self-assembly dates back to the 1980's, when Seeman engineered "tiles" from DNA strands that could connect to other tiles [16]. Winfree [20] (and later Rothemund as well [13]) designed a Tile Assembly Model as a mathematical abstraction of nanotile self-assembly. This Tile Assembly Model has become a fundamental tool in both theoretical and practical research. Our focus, though, in this paper, is on a theoretical advance that came out of the field of robotics: *graph assembly systems*, introduced by Klavins *et al.* in 2004 [8]. Graph assembly systems are a special class of graph grammars, and they provide a symbolic and topological characterization of a wide variety of systems of self-assembly. For example, a graph assembly system that simulates the Tile Assembly Model appears in [6]. It turns out that graph assembly systems are a "dual notion"

1

of a graph rewriting characterization of distributed systems that was proposed by Degano and Montanari over twenty years ago [2]. We explore that observation in this paper, so the theoretical questions of self-assembly—such as management of fault tolerance—can benefit from thirty years of upper and lower bounds in distributed computing.

Graph grammars were introduced in the 1970's, and are a generalization of classical string grammars used in automata theory [3]. They have been used in compression algorithms, studies of software architecture, data structure maintenance, and many other applications [15] [4]. Graph grammars are defined by a set of "left-hand" rules and "right-hand" rules. Given a starting graph G, if G contains one of the left-hand rules as a subgraph $H \subseteq G$, then G is redrawn as G', with the corresponding right-hand rule now appearing where H used to be. Graph assembly systems restrict the left-hand and right-hand rules so that each rule set operates on the same cardinality of nodes. Intuitively, if there are self-assembling particles in solution, the left-hand rule represents the particles in a starting conformation; and the right-hand rule, the resulting conformation, once the assembly step has been applied. Each graph assembly system includes a labeling function. Intuitively, each node in the graph represents a self-assembling agent, and the node's label denotes its current state.

Articles tend to use "graph grammar" when emphasizing the properties of the rules, and "graph rewriting system" when emphasizing how to apply the rules and draw the resulting graphs. Degano and Montanari presented a graph rewriting system, intending to provide a visualization of how distributed systems evolved over time. Nodes in their graphs were either ports or processes, and processes communicated with each other through ports. Subsystems of processes were defined with hyperedges (so their graphs were actually hypergraphs), and the left-hand and right-hand writing rules could contain nodes, node labels, edges and hyperedges. They defined an ultrametric space over the possible computations of a distributed system, and proved theorems about, for example, the relation between fair and terminal computations. We translate graph assembly systems into this framework by considering each self-assembling agent as a process, each edge between agents as embedding a port, and subsystems as agents binding through ports.

Several researchers have expressed the sentiment that self-assembly and distributed computing are related. Klavins, in particular, reported programming self-assembling robots with graph assembly systems, and called graph grammars "distributed algorithms" [7]. However, the application of distributed computing impossibility results to the field of self-assembly is quite recent. The first such application appeared in [19], and a tile assembly simulation of the consensus problem for some systems of distributed processes appeared in [18].

The rest of the paper is structured as follows. Section 2 provides background on graph assembly systems, and graph grammars for distributed systems. Section 3 proves that graph assembly systems are a "dual notion" of grammars for distributed systems. In Section 4 we prove a strengthening of Soloveichik and Winfree's theorem on local determinism [17] (a sufficient condition for a tile assembly system to produce a unique terminal assembly in the Tile Assembly Model). In Section 5 we present a canonical way to simulate distributed computing results with self-assembly. Section 6 concludes the paper and suggests directions for future research.

2 Background

2.1 Graph assembly systems

In this subsection, we provide the basic definitions of graph assembly systems. We refer the reader to [6] for a high-level explanation with several motivating examples.

All graphs in this subsection are simple labeled graphs over a finite alphabet Σ. Later in the paper we will extend these graphs to hypergraphs. For now, a *graph* $G = (V, E, l)$ is a triple, where V is a set of vertices, E a set of edges, and $l : V \to \Sigma$ a labeling function. If G is a graph, we sometimes write V_G and E_G to denote the vertices, or edges, of G, respectively. A *rule* is a pair of graphs $r = (L, R)$ where $V_L = V_R$. L is called the *left-hand side of r*, and R, the *right-hand side of r*.

Let G_1 and G_2 be graphs. A function h from V_{G_1} to V_{G_2} (often written $h : G_1 \to G_2$) is a *label preserving embedding* if (1) h is injective; (2) $\{x, y\} \in E_{G_1} \Rightarrow \{h(x), h(y)\} \in E_{G_2}$; (3) $l_{G_1} = l_{G_2} \circ h$. A rule *$r$ is applicable to a graph G* if there exists an embedding $h : L \to G$. An *action* on a graph G is a pair (r, h) such that r is applicable to G as witnessed by embedding h.

Definition 1. Let $G = (V, E, l)$ be a graph, $r = (L, R)$ a rule applicable to G, and (r, h) an action. The *application of (r, h) to G* produces a new graph $G' = (V', E', l')$, defined as follows.

$$V' = V$$
$$E' = [E \setminus \{\{h(x), h(y)\} \mid \{x, y\} \in L\}] \cup \{\{h(x), h(y)\} \mid \{x, y\} \in R\}$$
$$l'(x) = \begin{cases} l(x) \text{ if } x \notin h(V_L) \\ l_R \circ h^{-1}(x) \text{ otherwise} \end{cases}$$

Definition 2. A *graph assembly system* is a pair (G_0, Φ), where G_0 is the *initial graph* and Φ is a (finite) set of rules, called the *rule set*.

Intuitively, G_0 represents the initial configuration of self-assembling agents, before any binding rules have been applied; while Φ characterizes the binding rules of the system. Figures 1(i) and 1(ii) illustrate an example that first appeared in [6]: G_0 is a graph with no edges and a countably infinite vertex set, where each vertex is labeled "a"; and Φ, the rule set, is defined as

$$\Phi = \begin{cases} a & a & \Rightarrow & b - b \\ a & b & \Rightarrow & b - c \\ b & b & \Rightarrow & c - c. \end{cases}$$

We now define the notion of a language generated by a graph assembly system.

Definition 3. Let $G = (G_0, \Phi)$ be a graph assembly system. A connected graph H is *reachable* in G if there exists a sequence of rules in Φ that can be applied to G_0 in order to produce H. We write $\mathcal{R}(G_0, \Phi)$ for the set of graphs reachable in G. A connected graph $H \subseteq \mathcal{R}(G_0, \Phi)$ is *stable* if there exists no rule in Φ that can be applied to H. The *language* of G, written $\mathcal{L}(G)$, is the set of stable graphs that are reachable in G.

2.2 Grammars for distributed systems

In 1987, Degano and Montanari proposed a characterization of distributed systems based on graph rewriting [2], which they called "Grammars for Distributed Systems," or GDS. While their article included graph grammars, the focus was on the actual production of graphs that represented computations in distributed systems. Their work has not found broad application, due in part to the difficulty of implementing a graph rewriting system, which in general requires the solution of the NP-complete Subgraph Isomorphism Problem. However, their work has been used to characterize software architecture styles [11] [5].

We will present a limited version of GDS; we follow [2] closely in places. The original journal article by Degano and Montanari [2] is the best source for a full presentation.

An *alphabet of events and processes* $\Sigma = (T, M)$ is defined such that: each of T and M is a finite, or countably infinite, alphabet, such that each element of T (or M) is assigned a natural number by a ranking function. We call the elements of T *event names*, and the elements of M *process names*. Intuitively, the ranking function on Σ assigns a k-arity to each symbol that indicates how many ports are associated with the event, or connected to the process. (A port provides a physical connection between two separate processes.) A *hypergraph* $H = (N, S, f)$ is a triple where N is a set of nodes, S is a set of hyperarcs, and $f : S \rightarrow \bigcup_{k=1}^{\infty} N^k$ is a connection function that assigns to every hyperarc a k-tuple specifying the nodes to which it is connected. Two or more hyperarcs sharing at least one node are called *adjacent*.

Definition 4. A *distributed system* on an alphabet of events and processes Σ is a 5-tuple $\mathcal{D} = (N, S, f, l, \leq)$ such that

1. (N, S, f) is a hypergraph. We call the elements of S *subsystems*.

2. $l : S \rightarrow \Sigma$ is a labeling function such that, if $l(s) = x$, then the hyperarc s and the symbol x have the same rank. A subsystem labeled with a symbol from T is called an *event*. A subsystem labeled with a symbol from M is called a *process*.

3. $\leq$ is a partial ordering on S. If s_1 and s_2 are subsystems, and $s_1 \leq s_2$ or $s_2 \leq s_1$, we say s_1 and s_2 are *causally related*. If s_1 and s_2 are not causally related, they are *concurrent*. We require that all $\leq$-predecessors of events are themselves events. We also require that no more than two concurrent subsystems can be adjacent, and an event cannot be concurrent with any adjacent subsystem.

A distributed system with a finite number of subsystems is called *finite*. A distributed system is *rooted* if it contains exactly one event, and that event is $\leq$-smaller than any other hyperarc in the system. We now define a notion of graph "productions," or building blocks of rewriting rules, which describe the evolution of subsystems as isolated entities.

Definition 5. A *production* $p = (X^k, (\mathcal{D}, (n_1, \ldots, n_k)))$ of rank k on the alphabet of events and processes Σ, is a pair whose left element is a member of Σ with rank k, and whose right element is a pair $(\mathcal{D}, (n_1, \ldots, n_k))$, where $\mathcal{D} = (N, S, f, l, \leq)$ is a distributed system on Σ and $(n_1, \ldots, n_k)$ is a k-tuple of *distinct* nodes of $\mathcal{D}$.

The nodes $(n_1, \ldots, n_k)$ are called *external*, and the other nodes of N are called *internal*. Intuitively, the external nodes are the nodes of the production that can communicate with (*i.e.*, be a part of) other productions, so the productions can combine to form more complex rewriting rules. We formalize the construction of rewriting rules from productions as follows.

Definition 6. A *rewriting rule* $r = (\mathcal{D}_1, (\mathcal{D}_2, g, R))$ is a pair whose left member is a distributed system $\mathcal{D}_1 = (N_1, S_1, f_1, l_1, \leq_1)$. We require that the hypergraph (N_1, S_1, f_1) be connected; that all subsystems of $\mathcal{D}_1$ are labeled with process names, not event names; and that all subsystems of $\mathcal{D}_1$ be pairwise concurrent. The right member $(\mathcal{D}_2, g, R)$ is a triple such that

1. $\mathcal{D}_2 = (N_2, S_2, f_2, l_2, \leq_2)$ is a rooted distributed system. Let e be the unique event in S_2.

2. $g : N_1 \to N_2$ is an injective function, called the *spatial embedding function*. For every $n \in N_1$, we require that, if n is connected to two processes, then the node $g(n)$ is connected to event e; otherwise, node $g(n)$ cannot be connected to event e, and two subsystems both connected to node $g(n)$ cannot be concurrent.

3. $R \subseteq S_1 \times S_2$ is the *temporal embedding relation*. We require that R be such that sRx and $x' \leq_2 x$ imply sRx'. Furthermore, for all $s \in S_1$ we must have sRe. Finally, if $s' \in S_2$ is connected to a node $g(n)$, then there is an $s \in S_1$ such that sRs' and s is connected to n.

The purpose of g and R is to specify how, when applying the rewriting rule, the hypergraph in its right member can be properly inserted into the original hypergraph. In particular, g specifies the correspondence of the ports, and R of the subsystems in the right and left member.

Definition 7. A *grammar for a distributed system (GDS)* is a triple $G = (\Sigma, \mathcal{D}_0, P)$, where Σ is the alphabet of events and processes, $\mathcal{D}_0$ is an initial finite distributed system with no events, and P is a set of productions. Given two distributed systems $\mathcal{D}$ and $\mathcal{D}'$ on Σ, we write $\mathcal{D} \xrightarrow{G} \mathcal{D}'$ iff there exists a rewriting rule derivable from P that transforms $\mathcal{D}$ into $\mathcal{D}'$.

Definition 8. A *computation* for GDS G is a (finite or infinite) sequence $\{\mathcal{D}_i\} = \langle \mathcal{D}_0, \mathcal{D}_1, \ldots \rangle$ such that $\mathcal{D}_i \xrightarrow{G} \mathcal{D}_{i+1}, i = 0, 1, \ldots$. A distributed system $\mathcal{D}$ is *final* if there exists no $\mathcal{D}'$ such that $\mathcal{D} \xrightarrow{G} \mathcal{D}'$. A computation is *successful* iff its result is final. A computation is *weakly fair* iff any process to which a production can be applied will eventually have some production applied to it. The *language* $\mathcal{L}(G)$ *generated by* G is the set of the distributed systems that are the results of all successful computations of G.

Degano and Montanari intended GDS to draw graphs from top to bottom, depicting the temporal flow of distributed computations as one moved down the page. We omit their procedures for applying rewriting rules, because our interest is not in graph drawing, but in the theoretical results they obtained regarding an ultrametric space of GDS computations. We present that now.

Recall that an *ultrametric* on a set I is a function $d : I \times I \to \mathbb{R}_+$ that is reflexive, symmetric, and satisfies the condition $d(x, z) \leq \max\{d(x, y), d(y, z)\}$. If d is an ultrametric on I, then (I, d) is called an *ultrametric space*.

Given a distributed system $\mathcal{D} = (N, S, f, l, \leq)$ and a subsystem $s \in S$, let $\mathrm{depth}(s)$ be the natural number defined as the cardinality of a longest chain (without repetitions) consisting of $\leq$-predecessors of s. Then for any distributed system $\mathcal{D}$ and natural number n, we can define the *truncation* $[\mathcal{D}]_n$ of $\mathcal{D}$ at depth n by $[\mathcal{D}]_n = (N', S', f', l', \leq')$, where

$$S' = [S]_n = \{s \in S \mid \mathrm{depth}(s) < n\}$$
$$N' = \{n \mid (\exists s \in S)(\exists i \in \mathbb{N}) f(s)|_i = n\}$$
$$f', l', \leq' \text{ are the restrictions of } f, l, \leq \text{ to } S' \text{ respectively.}$$

Definition 9. Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be distributed systems. The *distance* $d(\mathcal{D}_1, \mathcal{D}_2)$ is defined as

$$d(\mathcal{D}_1, \mathcal{D}_2) = \begin{cases} 2^{-\max\{n \mid [\mathcal{D}_1]_n = [\mathcal{D}_2]_n\}} & \text{if such a maximum exists} \\ 0 & \text{otherwise.} \end{cases}$$

Let $\mathfrak{D}$ be the set of all distributed systems in which each subsystem has a finite number of predecessors, and in which only finitely many concurrent steps occur simultaneously. Let $\mathrm{Fin}(\mathfrak{D})$ be the set of all finite distributed systems in $\mathfrak{D}$. Degano and Montanari proved the following.

Theorem 1. $(\mathfrak{D}, d)$ is an ultrametric space. Further, $(\mathfrak{D}, d)$ is the completion of $(\mathrm{Fin}(\mathfrak{D}), d)$. Every infinite computation $\{\mathcal{D}_i\}$ is a (convergent) Cauchy sequence in $(\mathfrak{D}, d)$.

If a computation is finite, its *result* is the last element of the computation. If the computation is infinite, its result is its limit in $(\mathfrak{D}, d)$. This limit is guaranteed to exist by Theorem 1. Because of this, Degano and Montanari were also able to prove the next theorem.

Theorem 2. An infinite computation is weakly fair iff it is successful.

3 Graph assembly system as distributed system

The fundamental observation of this paper is that a graph assembly system G is the "dual" of a GDS, in the following sense: the edges of graphs derived from G correspond to ports (nodes) in a GDS, and the vertices in graphs derived from G correspond to processes (hyperedges) in a GDS. We formalize this observation with the next theorem.

Theorem 3. Let $G = (G_0, \Phi)$ be a graph assembly system. Then there is a grammar for a distributed system $G^* = (\Sigma, \mathcal{D}_0, P)$ and an injective mapping $\psi : \mathcal{R}(G_0, \Phi) \to (\mathfrak{D}, d)$ such that

1. $\psi(G_0) = \mathcal{D}_0$.

2. $H \in \mathcal{L}(G) \Leftrightarrow \psi(H) \in \mathcal{L}(G^*)$.

3. If $\langle G_0, H_1, H_2, \ldots \rangle$ is a (finite or infinite) sequence such that each graph in the sequence can be obtained by applying rules from Φ to the graph preceding it in the sequence, then $\langle \mathcal{D}_0, \psi(H_1), \psi(H_2), \ldots \rangle$ is a legal computation in G^*.

Proof. Fix graph assembly system $G = (G_0, \Phi)$. We construct ψ and G^* as follows.

We assume Φ is finite, though G_0 may be infinite. Let Σ be the alphabet of G. We define Σ^*, the alphabet of events and processes for GDS G^* by: (1) For each L such that $(\exists R)[(L, R) \in \Phi]$, place a unique event name L^* in Σ^*; (2) for each vertex label $\lambda \in \Sigma$, place a unique process name λ^* in Σ^*. This constructs $\Sigma^* = (T, M)$.

Let k be the maximum degree of a node in any L or R such that $(L, R) \in \Phi$, and fix an orientation on each rule in Φ so the edges of each node are marked first, second, third, up to k-th. For node v with label $\lambda \in \Sigma$, define $\psi(v) \triangleq s$, where s is a hyperedge on k nodes with label $\lambda^* \in \Sigma^*$. For nodes u, v with an edge e between them, let i, j be the orientation markers of where the edge connects to u, v. Define $\psi(e) \triangleq e'$, where e' is an edge from the i-th port of $\psi(u)$ to the j-th port of $\psi(v)$.

Define $G^* = (\Sigma^*, \mathcal{D}_0, P)$ so $\mathcal{D}_0 \triangleq \psi(G_0)$, and $P \triangleq \{(\psi(L), L^* \mapsto \psi(R)) \mid (L, R) \in \Phi\}$, where $L^* \mapsto \psi(R)$ is the subgraph obtained by drawing an edge from the event L^* to the subsystem

$\psi(R)$. Then $\psi(G_0) = \mathcal{D}_0$, and, if H is reachable from G_0, then $\psi(H)$ can be generated from G^* by applying the appropriate rule, translated from Φ to P. So any legal application of rules of G induces a legal computation in G^* via application of ψ. Finally, since we defined ψ injectively, $H \in \mathcal{L}(G) \Leftrightarrow \psi(H) \in \mathcal{L}(G^*)$. $\qquad\square$

Figure 1 shows a GDS representation of the behavior of the graph assembly system that appeared in Section 2.1.

4   Generalization of local determinism

In this section, we prove a generalized version of a theorem about tile self-assembly due to Soloveichik and Winfree. They were interested in guaranteeing that a tile assembly system would always form a unique terminal assembly. For reasons of space, we defer the formal definition of the Winfree-Rothemund Tile Assembly Model—and a graph assembly system characterization of it—to the Appendix. Here we use the ultrametric space of legal GDS computations to provide a sufficient condition for self-assembling a unique structure, and we obtain Soloveichik and Winfree's theorem as a corollary.

First, we define a generalized notion of local determinism.

Definition 10. Let $G = (\Sigma, \mathcal{D}_0, P)$ be a GDS. We say G is *locally deterministic* if the following holds for all computations $\{\mathcal{D}_i\}$ generated by G. For any $k > 1$ and any process $s \in \mathcal{D}_k$, let $\mathcal{D}_j$ be maximal such that $\mathcal{D}_j \in \{\mathcal{D}_i\}$ and $s \notin \mathcal{D}_j$. Then there is exactly one production applicable to the parents (*i.e.*, immediate $\leq$-predecessors) of s, and that production produces s in the location where it appears in $\mathcal{D}_k$.

In words, the initial graph and the productions of G are such that the local subsystems of any finite computation entirely determine their children at the future computation step when a production is applied to them. Because the space $(\mathfrak{D}, d)$ is complete, we obtain the next theorem.

Theorem 4. Let $G = (\Sigma, \mathcal{D}_0, P)$ be a GDS that is locally deterministic. Then all (finite or infinite) weakly fair computations generated by G have the same result.

Proof. If the successful result of a computation generated by G is finite, then suppose there are two finite computations $\{\mathcal{D}_i^1\}$ and $\{\mathcal{D}_i^2\}$, each producing a different result. Then there must be some n such that $[\mathcal{D}^1]_n = [\mathcal{D}^2]_n$ but $[\mathcal{D}^1]_{n+1} \neq [\mathcal{D}^2]_{n+1}$. Let s be a process that witnesses the difference; suppose WLOG that $s \in [\mathcal{D}^1]_{n+1}$. Since G is locally deterministic, and both computations are weakly fair, s will eventually appear, with exactly the same predecessors, in $\{\mathcal{D}^2\}$. So the results of the two computations will eventually be equal.

Now suppose G generates two infinite weakly fair computation $\{\mathcal{D}_i^1\}$ and $\{\mathcal{D}_i^2\}$. By Theorem 2, both $\{\mathcal{D}_i^1\}$ and $\{\mathcal{D}_i^2\}$ are successful. But then we can argue as above. If the results of the two computations differ, they differ at some subsystem with finitely many predecessors. As those predecessors determine their child to be the unique subsystem that supposedly causes the difference, the assumption contradicts local determinism. $\qquad\square$

As graph assembly systems—and GDS's—can simulate a wide range of self-assembly models, including Winfree-Rothemund tile self-assembly, Soloveichik and Winfree's theorem is a corollary to Theorem 4.
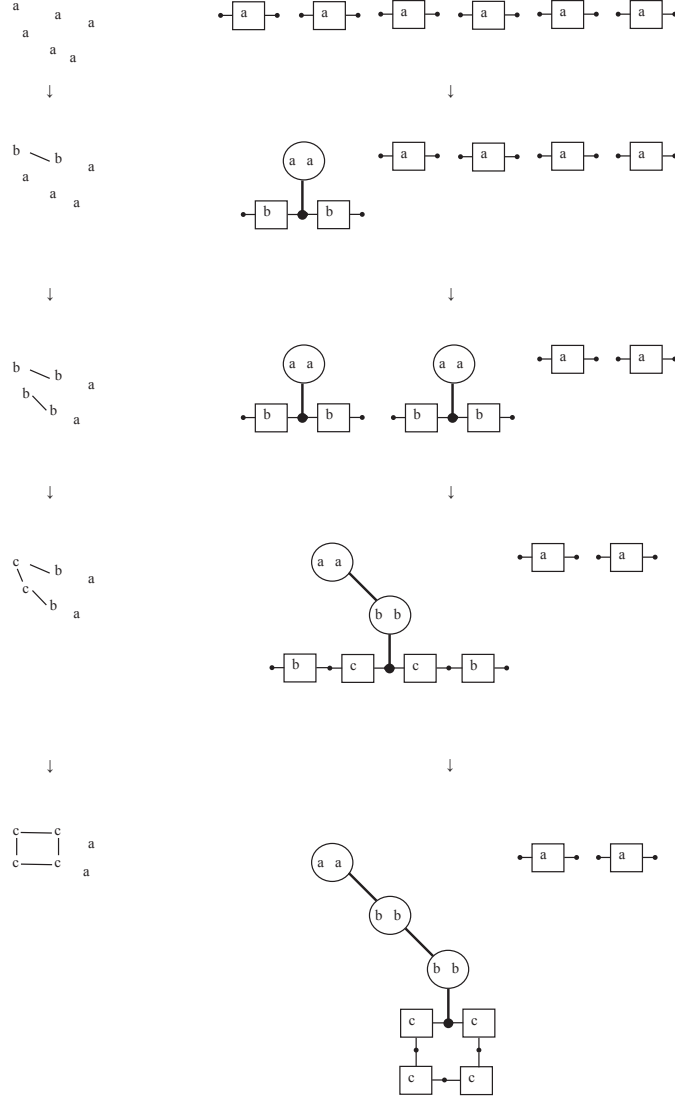
7

Figure 1: A comparison of a graph assembly system and its GDS representation. This figure is based on Example 1 in [6], which is also the example of a graph assembly system that appears in Section 2.1 of this paper. In the column on the left, we see one possible assembly sequence from the initial graph and rule set. On the right, we see the same behavior represented in GDS format.

Corollary 1 (Soloveichik and Winfree [17])**.** Let $\mathcal{T} = (T, \sigma, \Sigma, \tau, R)$ be a tile assembly system that is locally deterministic (in the sense of tile self-assembly, as defined in the Appendix). Then $\mathcal{T}$ has a unique terminal assembly.

We defer the proof to the Appendix, but the idea is that $\mathcal{T}$ can be simulated by a locally deterministic GDS, which will have a unique result per Theorem 4.

We conclude this section by noting that the "converse" question—given a terminal assembly $\mathcal{A}$, construct a graph grammar whose unique result is $\mathcal{A}$—is not well understood. An early complexity result in tile self-assembly showed that finding a *minimal* tileset that uniquely produces an assembly is NP-complete [1]. More generally, unless P=NP, there is no polynomial-time compression algorithm such that, given a string, the algorithm produces a grammar whose size is within a small constant factor of the minimal grammar that produces the string [10]. In the positive, Klavins *et al.* have proposed an algorithm that, given an arbitrary graph G, constructs a graph grammar whose unique result is G [8]. Their algorithm has not been implemented, but Peshkin has reported implementation of a graph grammar compression algorithm to analyze DNA molecules [12]. The approximation ratios obtained by these algorithms are unknown.

5 Simulation of other distributed systems

Authors sometimes distinguish between *active* self-assembly, and *passive* self-assembly. Active self-assembling agents (like some robots) can send neighbors messages after initially binding, and can decide to dissolve a bond after it has been made. Passive self-assembling agents (like molecules) just form an initial bond, and do not communicate with their neighbors once the bond has formed. Graph grammars can represent both types of self-assembly system. The main objective of this section is to present a canonical method to simulate models of distributed computing with passive systems of self-assembly described by GDSes (hence active systems can perform the same simulations).

For the rest of this section, let $\mathcal{M}$ be an asynchronous message-passing model of distributed computing with n processors, such that each processor runs forever, and can send and receive messages of constant-size complexity. Intuitively, $\mathcal{M}$ can be simulated by a passive system of self-assembly if the logic of each processor can be simulated by a distinct subsystem, and the simulation of any message sent from processor p_i to processor p_j eventually arrives (with probability one) at the subsystem that is simulating p_j, and is incorporated into that subsystem's execution.

Definition 11. Let G be a graph assembly system, and G^* its induced GDS. Then we say G^* *simulates* $\mathcal{M}$ if:

1. There is a 1-1 mapping h from configurations of $\mathcal{M}$ to distributed systems derivable from G^*.

2. If $C_0, \phi_0, C_1, \phi_1 \cdots C_i, \phi_i$ is a legal execution segment of $\mathcal{M}$, then $h(C_0) \longrightarrow h(C_1) \longrightarrow \cdots \longrightarrow h(C_i)$ is a legal computation derivable from G^*.

3. If there is no legal execution segment from C_0 to C_1 in $\mathcal{M}$, then there is no legal computation derivable from G^* such that $h(C_0) \longrightarrow h(C_1)$.

4. Let C be a configuration of $\mathcal{M}$ and $\mathcal{C}$ be a set of configurations of $\mathcal{M}$. If $\mathcal{M}$ is such that, upon achieving configuration C, it must eventually achieve some configuration $C' \in \mathcal{C}$, then

G^* is such that, if it ever reaches $h(C)$ then it must achieve $h(C')$ for some $C' \in \mathcal{C}$. (Note that $\mathcal{C}$ may be an infinite set.)

5. If Φ is a legal, finite execution segment in $\mathcal{M}$ that contains the event that p_i places message m in the outbuffer intended for processor p_j, then, with probability one, all legal computations derivable from $h(\Phi)$ will include some $\mathcal{D}$ such that $h^{-1}(\mathcal{D})$ is a configuration of $\mathcal{M}$ in which m has arrived at p_j.

6. If in configuration C of $\mathcal{M}$ all processes have halted, then the distributed system $h(C)$ derivable from G^* is terminal.

Graph grammars—and graph assembly systems—are Turing universal. However, the weakness of the graph grammar characterization of self-assembly is that it captures only the topology, and not the geometric constraints of the system. This means that we can simulate any single processor (Turing machine, finite state machine, *etc.*) using self-assembling agents, but it may not be possible to simulate a network of distributed processors, because the communication between processors may interfere with the system's ability to grow, depending on how the agents embed themselves into their geometric environment. To provide a specific example, it is not possible for the Winfree-Rothemund Tile Assembly Model to simulate a 3-consensus object in two-dimensional tile assembly—though it can be done in three dimensions—because there is no way to ensure that three independently-growing subassemblies can have wait-free access to a common decision point [18].

Therefore, while the next theorem shows how GDSes can simulate distributed systems topologically, it does not guarantee that such simulations are achievable in practice, because the GDS may not be embeddable into the plane, or into three-space. We also present a theorem that provides a specific example of a simple distributed system that can be simulated by two-dimensional self-assembly. The general relationship between the topology and geometry of self-assembly is unknown.

Theorem 5. Let $\mathcal{M}$ be an asynchronous message-passing model of distributed computing such that all processors run forever, and each processor can send messages of size bounded by some constant k. Then there is a graph assembly system (and, hence, GDS) that simulates $\mathcal{M}$.

Proof. Let $p_1, \ldots, p_n$ be the processors of $\mathcal{M}$. Since graph assembly systems are Turing universal, there are graph assembly systems $G_1, \ldots, G_n$ that simulate $p_1, \ldots, p_n$. For simplicity, we will use, as a "base" for our construction, a simulation of the n processors by 4-regular self-assembling agents that can be embedded in the plane. Winfree showed (in his Tile Assembly Model) that any Turing machine could be simulated in this way [20], by a planar "wedge" construction. This wedge construction can be modified to simulate a processor sending a constant-size message to another location in the plane, and the angle at which the wedge grows can be modified to fit as many wedges as necessary into a single quadrant of the plane [9]. So if $\mathcal{M}$ is such that no processor ever sends a message to any other, we can build a GDS that simulates $\mathcal{M}$ easily, by starting with an initial graph G_0 that encodes wedges that simulate each p_i. As long as we are careful to ensure the rule set Φ constructs wedges that grow without colliding with one another—which we can always do—then (G_0, Φ) is a graph assembly system that simulates $\mathcal{M}$.

So assume that some processors in $\mathcal{M}$ send messages to one another. We will enhance the wedges that simulate the logic of each processor by defining agents that build along one side to act as an inbuffer, and along the other side to simulate the sending of a message from one processor

to another. In general, our construction cannot be embedded into two dimensions, and will require agents with more than four ports each. However, to visualize how the construction operates, it may help to review Figures 6, 7 and 8, which show how communication between two processors can be simulated in the Tile Assembly Model.

Let W_i be the graph grammar that produces the wedge that simulates p_i. WLOG, assume that W_i grows northward, and grows in width only to the west. We will simulate an inbuffer along the east side of W_i, and modify it so that infinitely many rows simulate checking the value of that inbuffer. Figure 2(i) and 2(ii) shows how to modify a wedge so that the information "message m received" or "no message received" can be incorporated into the overall logic of the wedge.

Let $X \subseteq \{(i,j) \mid 1 \leq i \neq j \leq n\}$ be the set of pairs of processors such that p_i could send a message to p_j under some legal execution of $\mathcal{M}$. For each $(i,j) \in X$, we add an "information highway" from W_i to W_j by adding ports to the inbuffer of Figure 2(ii); this is shown in Figure 2(iii). We then define one agent (and corresponding rules) for each message and each element of X, so the agent simulates "message m is in transit from p_i to p_j." As there are constant-many possible messages, and n processors, we only need to define finitely many such agents. As p_i may send a second message m_2 to p_j before a first message m_1 has been delivered, we add ports to the message-encoding agents that duplicate the "information highway." This permits the creation of a queue: messages from p_i to p_j will arrive in the order sent. This step of the construction is shown in Figure 2(iv).

We complete construction of the inbuffer simulation mechanism with agents and rules that behave according to Figure 3. If messages $m_1, \ldots, m_k$ from distinct processors $p_{i_1}, \ldots, p_{i_k}$ are waiting in the inbuffer, one of them will eventually enter the inbuffer with probability one. This is guaranteed because we are assuming that if more than one rule in Φ is applicable at a given time step, the rule that is applied will be chosen uniformly at random. Therefore, since it takes placement of only one agent to move a message up a wedge one row, but several tiles to build an entire wedge row, any message sent will (almost surely) eventually get to the top of the wedge. Note that this construction guarantees only that if m_1 and m_2 are sent by the same processor, and m_1 is sent first, then m_1 will enter the inbuffer before m_2. Messages sent by different processors have no such prioritization. Nevertheless, as the inbuffer is checked infinitely often, and there can only be k-many messages competing to enter the inbuffer at any time step (for some fixed k), each message waiting at a given time step s will eventually enter with probability one, satisfying our definition of simulation of an asynchronous system. (We are glossing over the mechanism that moves m_2 to the front of the p_i-queue, if p_i sent m_1 and m_2, and then m_1 entered the inbuffer. To construct this mechanism formally, we begin with a mechanism in the Tile Assembly Model that does this for a single processor: such a mechanism appears in Figure 8. We then define graph assembly system agents and rules for that mechanism, using Klavins' graph grammar simulation of the Tile Assembly Model presented in the Appendix.)

To complete the proof, we note that the simulation of the sending of messages is straightforward, and illustrated in Figure 4. Since we are only considering topology, and do not care about embedding the agents into a geometric space, we can assume that each wedge grows in width by one agent at every other row, without worrying about collision with other parts of the construction. Then, since we know the exact pattern of wedge growth, we can define agent rules that move a sent message down the side of the wedge that is growing (*i.e.*, that the inbuffer is not on), and hardcode additional binding rules into G_0 that send the message along ports of G_0 on to its destination wedge, bringing us back to the mechanism in Figure 3.

As we can build a graph assembly system that simulates $\mathcal{M}$, by Theorem 3, we can build a GDS that simulates $\mathcal{M}$ also. □

The above result is computability-theoretic, as it demonstrates possibility. We do not know the answers to complexity-theoretic questions like, "What simulations are possible with d-regular agents?" or, "What simulations are possible with agents embedded in the plane?" However, we are able to demonstrate that 4-regular agents embedded in the plane can simulate a two-processor message-passing model.

Theorem 6. For any two-processor message-passing model $\mathcal{M}$ of distributed computing in which each process runs forever and sends messages of constant size, there is a tile assembly system $\mathcal{T}$ (in the standard Winfree-Rothemund Tile Assembly Model) that simulates $\mathcal{M}$ in two dimensions.

We defer the proof of Theorem 6 to the Appendix.

6   Conclusion

We have shown how the graph grammar representation of networks of self-assembly can be naturally characterized, using Grammars for Distributed Systems (GDS). Use of GDS permits generalization of existing theorems about self-assembly. We also showed how to use graph assembly systems and GDSes to simulate asynchronous constant-size-message-passing models of distributed computing.

Perhaps the principal theoretical and practical barrier currently facing the field of self-assembly is the lack of management of fault tolerance. In order to import results about either crash failures or Byzantine failures from the world of distributed computing, we will need a better understanding of the relationship between what can be simulated topologically (*i.e.*, by GDSes) and what can be built when self-assembling agents are embedded into a particular geometric space. We see this relationship between topology and geometry as the primary open question to pursue.

References

[1] ADLEMAN, L., CHENG, Q., GOEL, A., HUANG, M.-D., KEMPE, D., DE ESPANÉS, P. M., AND ROTHEMUND, P. W. K. Combinatorial optimization problems in self-assembly. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing* (New York, NY, USA, 2002), ACM, pp. 23–32.

[2] DEGANO, P., AND MONTANARI, U. A model for distributed systems based on graph rewriting. *J. ACM 34*, 2 (1987), 411–449.

[3] EHRIG, H. Introduction to the algebraic theory of graph grammars (a survey). In *Proceedings of the International Workshop on Graph-Grammars and Their Application to Computer Science and Biology* (London, UK, 1979), Springer-Verlag, pp. 1–69.

[4] EHRIG, H., ENGELS, G., KREOWSKI, H.-J., AND ROZENBERG, G., Eds. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[5] HIRSCH, D., INVERARDI, P., AND MONTANARI, U. Graph grammars and constraint solving for software architecture styles. In *ISAW '98: Proceedings of the third international workshop on Software architecture* (New York, NY, USA, 1998), ACM, pp. 69–72.

[6] KLAVINS, E. Directed self-assembly using graph grammars. In *Foundations of Nanoscience: Self Assembled Architectures and Devices, Snowbird, UT* (2004).

[7] KLAVINS, E. Programmable self-assembly. *Control Systems Magazine 24*, 4 (August 2007), 43–56.

[8] KLAVINS, E., GHRIST, R., AND LIPSKY, D. Graph grammars for self assembling robotic systems. In *ICRA* (2004), IEEE, pp. 5293–5300.

[9] LATHROP, J. I., LUTZ, J. H., PATITZ, M. J., AND SUMMERS, S. M. Computability and complexity in self-assembly. In *CiE* (2008), A. Beckmann, C. Dimitracopoulos, and B. Löwe, Eds., vol. 5028 of *Lecture Notes in Computer Science*, Springer, pp. 349–358.

[10] LEHMAN, E., AND SHELAT, A. Approximation algorithms for grammar-based compression. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2002), Society for Industrial and Applied Mathematics, pp. 205–212.

[11] MÉTAYER, D. L. Software architecture styles as graph grammars. *SIGSOFT Softw. Eng. Notes 21*, 6 (1996), 15–23.

[12] PESHKIN, L. Structure induction by lossless graph compression. In *DCC* (2007), IEEE Computer Society, pp. 53–62.

[13] ROTHEMUND, P. W. K. *Theory and Experiments in Algorithmic Self-Assembly*. PhD thesis, University of Southern California, 2001.

[14] ROTHEMUND, P. W. K., AND WINFREE, E. The program-size complexity of self-assembled squares (extended abstract). In *STOC* (2000), pp. 459–468.

[15] ROZENBERG, G., Ed. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[16] SEEMAN, N. C. De novo design of sequences for nucleic acid structural engineering. *Journal of biomolecular structure and dynamics 8*, 3 (1990), 573–581.

[17] SOLOVEICHIK, D., AND WINFREE, E. Complexity of self-assembled shapes. *SIAM J. Comput. 36*, 6 (2007), 1544–1569.

[18] STERLING, A. Distributed agreement in tile self-assembly. In *DNA 15* (June 2009). To appear.

[19] STERLING, A. D. A limit to the power of multiple nucleation in self-assembly. In *DISC* (2008), G. Taubenfeld, Ed., vol. 5218 of *Lecture Notes in Computer Science*, Springer, pp. 451–465.

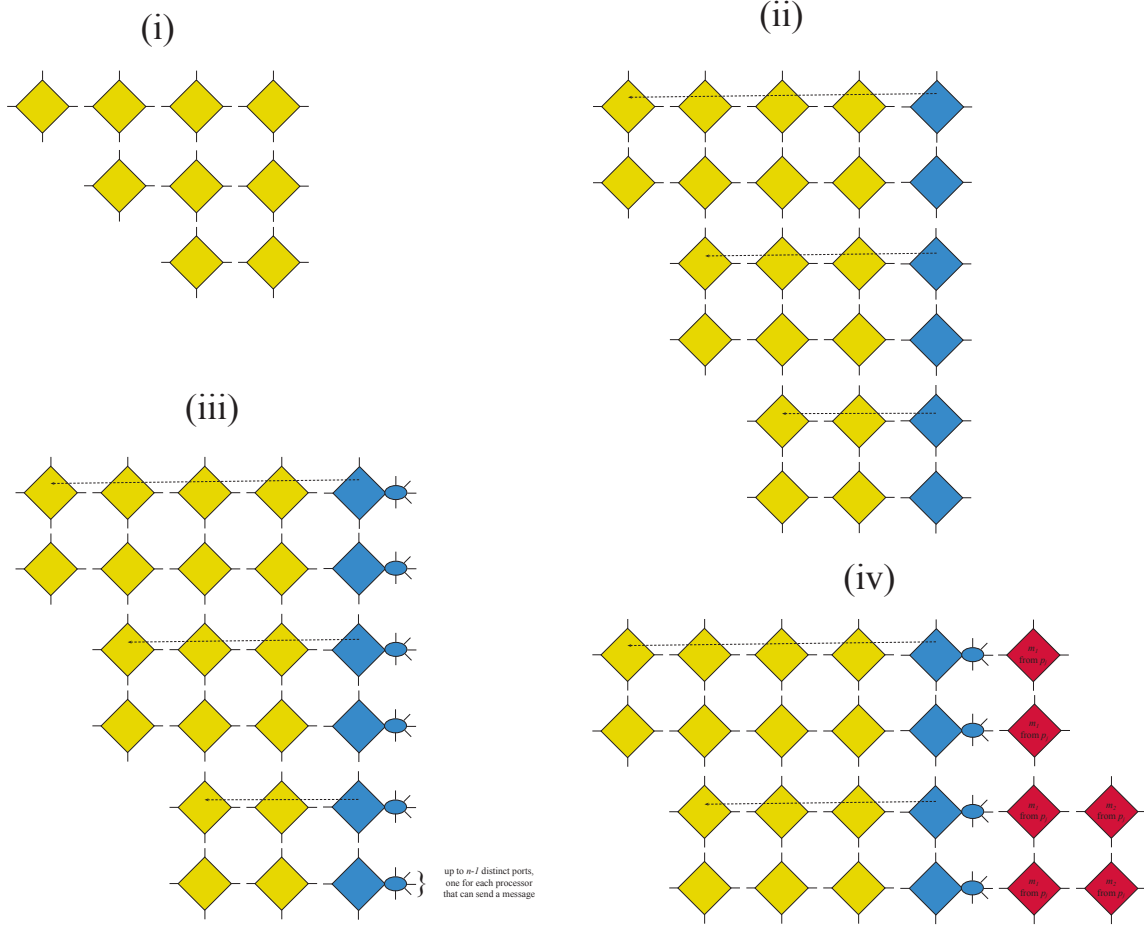[20] WINFREE, E. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, 1998.

Figure 2: This figure illustrates how to simulate a processor's inbuffer, given a "wedge" construction (symbolized by the yellow tiles) that simulates the logic of the processor, as in (i). In stage (ii), we interleave the logic of the processor with an "inbuffer checking row," such that the agents only bind once the blue inbuffer agent has bound to the structure. The agents in the inbuffer row copy information from south to north, so no information is lost in the construction. The dashed arrows show the direction in which agents bind at inbuffer rows. In stage (iii), we modify the inbuffer agents so they have unique ports for each processor that might potentially send a message to the processor the yellow agents are simulating. Stage (iv) shows this in action: processor p_j has sent messages m_1 and m_2, in that order, and both messages are "climbing" up the edge of the wedge.
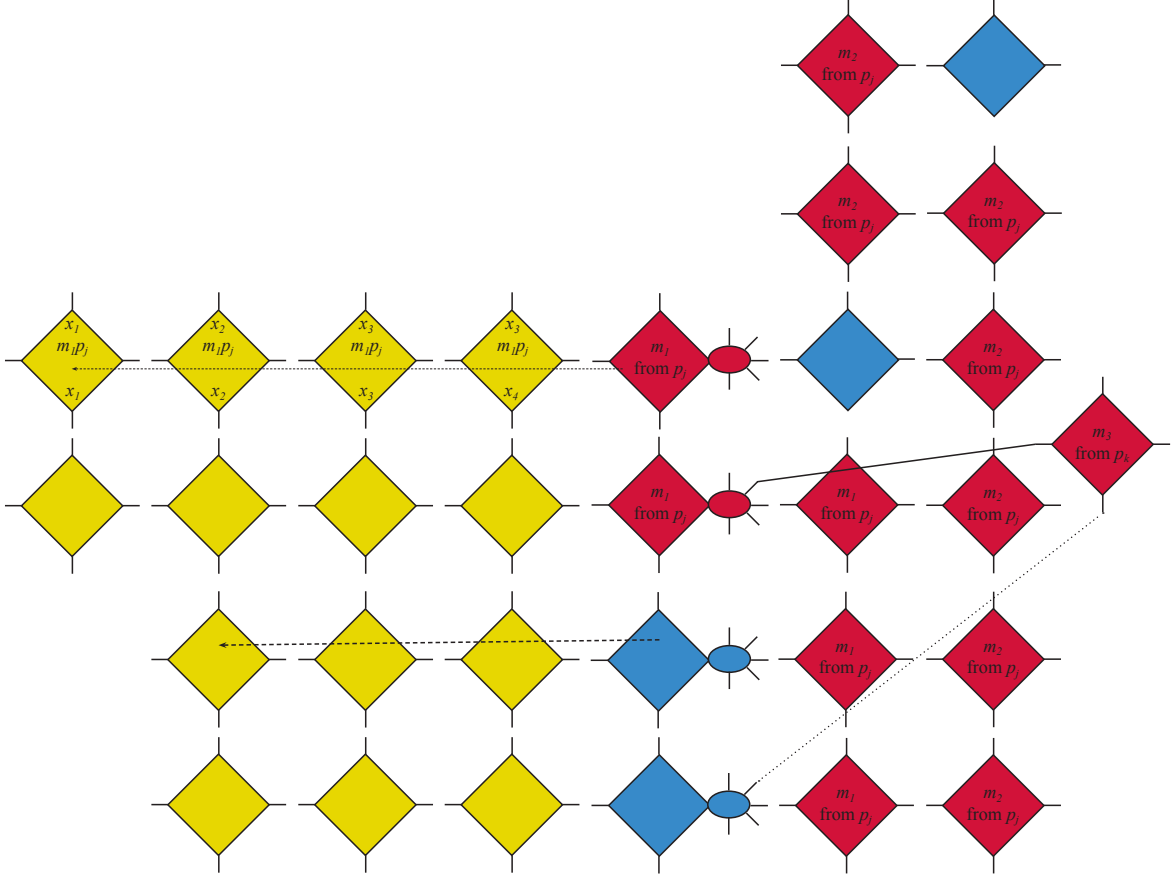
14

Figure 3: The conclusion of the inbuffer simulation described in Figure 2. We see three messages waiting: m_1 and m_2, both sent from processor p_j; and m_3, sent from processor p_k. The predecessors of the agent that simulates m_3 are connected to the appropriate ports along the inbuffer, like the agents that carried m_1, but we have represented them with a dotted line, for clarity. Three possible agents can bind to the inbuffer location at the northeast corner of the wedge: an agent (like before) that simulates no message received yet; an agent simulating that m_1 was received, and an agent simulating the receipt of m_3. (Note that m_2 cannot be received until m_1 has been delivered. Then it will be advanced to the head of the queue using a mechanism similar to the one used in tile assembly as shown in Figure 8.)
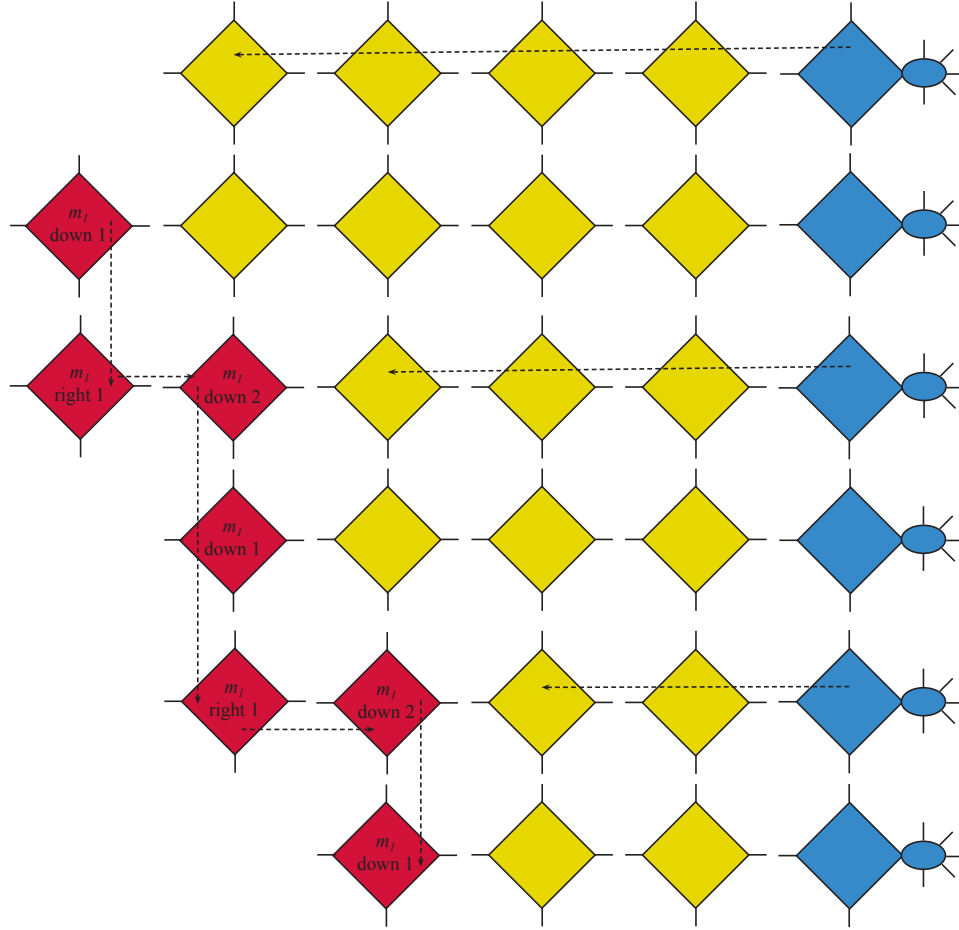
Figure 4: As the yellow wedge grows, if the processor logic indicates a message be sent, the wedge sends a tile encoding the message, and the direction and distance it travels, down the edge of the wedge that is not in use by the inbuffer. This is represented by the red-colored tiles. The ports on the west side of the red tiles are available in case the wedge wants to send another message at a future point in the construction.

A   Winfree-Rothemund Tile Assembly Model

Winfree's objective in defining the Tile Assembly Model was to provide a useful mathematical abstraction of DNA tiles combining in solution in a random, nondeterministic, asynchronous manner [20]. Rothemund [13], and Rothemund and Winfree [14], extended the original definition of the model. For a comprehensive introduction to tile assembly, we refer the reader to [13]. Intuitively, we desire a formalism that models the placement of square tiles on the integer plane, one at a time, such that each new tile placed binds to the tiles already there, according to specific rules. Tiles have four sides (often referred to as north, south, east and west) and exactly one orientation, $i.e.$, they cannot be rotated.

A tile assembly system $\mathcal{T}$ is a 5-tuple $(T, \sigma, \Sigma, \tau, R)$, where T is a finite set of tile types; σ is the $seed\ tile$ or $seed\ assembly$, the "starting configuration" for assemblies of $\mathcal{T}$; $\tau : T \times \{N, S, E, W\} \rightarrow \Sigma \times \{0, 1, 2\}$ is an assignment of symbols ("glue names") and a "glue strength" (0, 1, or 2) to the north, south, east and west sides of each tile; and a symmetric relation $R \subseteq \Sigma \times \Sigma$ that specifies which glues can bind with nonzero strength. In this model, there are no negative glue strengths, $i.e.$, two tiles cannot repel each other.

A $configuration\ of\ \mathcal{T}$ is a set of tiles, all of which are tile types from $\mathcal{T}$, that have been placed in the plane, and the configuration is $stable$ if the binding strength (from τ and R in $\mathcal{T}$) at every possible cut is at least 2. An $assembly\ sequence$ is a sequence of single-tile additions to the frontier of the assembly constructed at the previous stage. Assembly sequences can be finite or infinite in length. The $result$ of assembly sequence $\overrightarrow{\alpha}$ is the union of the tile configurations obtained at every finite stage of $\overrightarrow{\alpha}$. The $assemblies\ produced\ by\ \mathcal{T}$ is the set of all stable assemblies that can be built by starting from the seed assembly of $\mathcal{T}$ and legally adding tiles. If α and β are configurations of $\mathcal{T}$, we write $\alpha \longrightarrow \beta$ if there is an assembly sequence that starts at α and produces β. An assembly of $\mathcal{T}$ is $terminal$ if no tiles can be stably added to it.

We are, of course, interested in being able to $prove$ that a certain tile assembly system always achieves a certain output. In [17], Soloveichik and Winfree presented a strong technique for this: local determinism. An assembly sequence $\overrightarrow{\alpha}$ is $locally\ deterministic$ if (1) each tile added in $\overrightarrow{\alpha}$ binds with the minimum strength required for binding; (2) if there is a tile of type t_0 at location l in the result of α, and t_0 and the immediate "OUT-neighbors" of t_0 are deleted from the result of $\overrightarrow{\alpha}$, then no other tile type in $\mathcal{T}$ can legally bind at l; the result of $\overrightarrow{\alpha}$ is terminal. Local determinism is important because of the following result.

Theorem 7 (Soloveichik and Winfree [17]). If $\mathcal{T}$ is locally deterministic, then $\mathcal{T}$ has a unique terminal assembly.

Klavins has shown how to model a Tile Assembly System with a graph assembly system (Example 7 in [6]). We summarize his construction here, for completeness.

Let $\mathcal{T} = (T, \sigma, \Sigma, \tau, R)$ be a tile assembly system. We will construct a graph assembly system $G = (G_0, \Phi)$ that models it as follows. To model tile edges, we extend Σ to alphabet Σ^* of G, by adding, for every $a \in \Sigma$, new symbols $(N, a), (S, a), (E, a), (W, a)$ and $(N, a)', (S, a)', (E, a)', (W, a)'$ to Σ^*. Intuitively, the unprimed symbols indicate that a is a symbol at an unmatched (north, south, east or west) edge, while the primed symbols indicate that a is a symbol at an edge that is bound to another tile. We also add four new symbols to Σ^*: x, y, x' and y'. The symbols x and x' represent the "center" of a tile in either a state that is bound (unprimed) or unbound (primed) to the surface, and y, y' represent points of the surface that either have a tile bound (unprimed) to

them or not (primed). We also add to Σ^* the new symbols N, S, E and W. These symbols will be used to specify the orientation of the underlying grid.

The initial graph G_0 of G consists of an underlying grid, and an infinite supply of tiles of each type. We can specify the initial seed assembly σ by using symbols from Σ^* to define a grid that has specific tiles attached to it in finitely many places. (See [6] for a depiction of a grid that represents the assembly surface.) The binding rules of $\mathcal{T}$ can be translated directly into assembly rules of G, but, in addition, we must add additional rules to ensure that all legal G-assemblies are planar. These rules appear in Figure 5.

B   Proof of Corollary 1

We now prove Soloveichik and Winfree's theorem that a locally deterministic tileset has a unique terminal assembly (*i.e.*, Corollary 1 in Section 4).

Proof of Corollary 1. Let $\mathcal{T}$ be a locally deterministic tile assembly system. Let $G_{\mathcal{T}}$ be a graph assembly system that simulates the behavior of $\mathcal{T}$, constructed as described in Section A, and let $G_{\mathcal{T}}^*$ be a GDS that simulates $G_{\mathcal{T}}$, as provided by Theorem 3. In the Winfree-Rothemund Tile Assembly Model, all tile assembly sequences are assumed to be fair, so any computation generated by $G_{\mathcal{T}}^*$ will be weakly fair. Note that if $\mathcal{T}$ is locally deterministic (in the sense of tile assembly), then $G_{\mathcal{T}}^*$ is locally deterministic (in the sense of this paper), because the predecessors of any tile t determine that t is the unique tile that can be placed at that location. Therefore, by Theorem 4, all computations generated by $G_{\mathcal{T}}^*$ will have the same result, which implies that $\mathcal{T}$ itself must have a unique terminal assembly. $\qquad\qquad\square$

C   Proof of Theorem 6

Proof. Fix $\mathcal{M}$ a two-processor asynchronous message-passing model of distributed computing, such that processors p_1 and p_2 send each other messages of size bounded by some constant k. Winfree has shown that the Tile Assembly Model is Turing Universal, and in particular can simulate a Turing Machine using a construction that grows in the shape of a "wedge" [20]. If neither p_1 nor p_2 ever sends (or receives) messages, then let $\mathcal{T}_1$ and $\mathcal{T}_2$ be tile assembly systems that simulate p_1 and p_2. Then, similar to the argument in the proof of Theorem 5, we can construct a tile assembly system $\mathcal{T}^*$ with a seed out of which $\mathcal{T}_1$ grows in one direction, and $\mathcal{T}_2$ grows in another. (To argue more formally, we could invoke the Multiseed Lemma of [9], which states that the terminal assembly of a tile assembly system composed of distinct "pseudoseeds" is, as expected, the union of the terminal assemblies of the individual pseudoseeds, as long as the assemblies do not physically interfere with one another as they grow.)

So assume that at least one of p_1, p_2 sends messages to the other. To simulate processors checking their inbuffers, we modify the wedge construction, so that (1) messages sent to the processor simulated by the wedge can "crawl up" the side of the wedge until they reach the frontier; and (2) every other row of tiles in the wedge allows sent messages to bind with that row with probability bounded away from zero. Therefore, the construction remains two-dimensional, and, since both p_1 and p_2 run forever, the message will be delivered with probability one. Figures 7 and 8 illustrate how the modified wedge construction works.
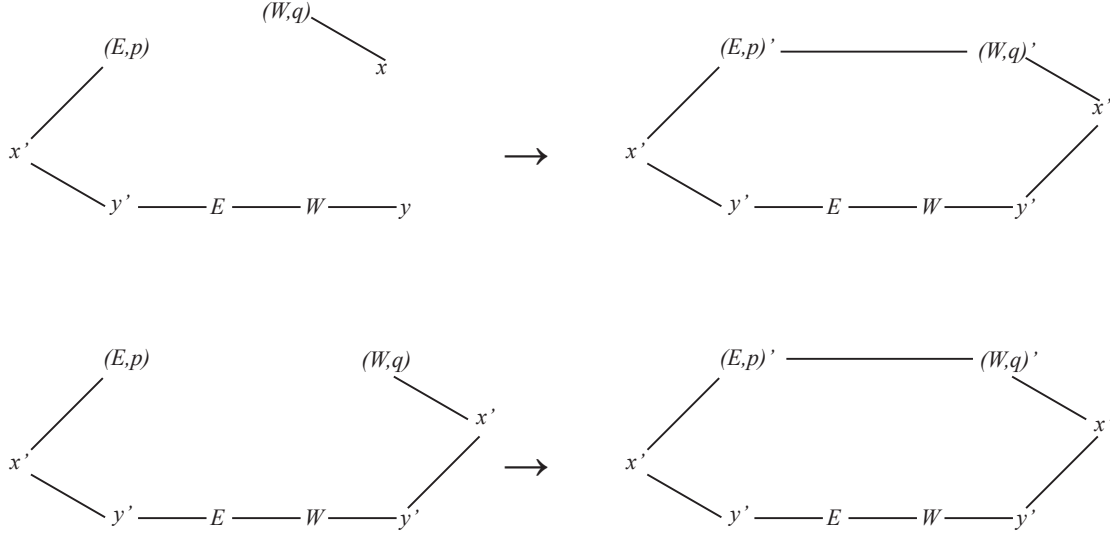
Figure 5: Binding rules to ensure planarity when simulating tile assembly systems with graph assembly systems (modeled after a figure in [6]). The top rule specifies how an unattached tile may bind to a tile that is already attached to the growing seed supertile. The bottom rule specifies how two adjacent tiles already bound to the seed supertile, but not to each other, may bind. We add two such rules for each $(p, q) \in R$, where R is the binding relation of the graph assembly system.

As with Theorem 5, since there are only k-many messages, we encode each message with a unique tile. We simulate the sending of messages from p_1 to p_2 by allowing tiles to bind along one "edge" of the tile assembly, and simulate the sending of messages from p_2 to p_1 along the other "edge." A high-level schematic for the construction appears in Figure 6.

It is important that the Tile Assembly Model assumes that at each stage of assembly, the tile that binds—and the location it binds to—are chosen uniformly at random. It takes several tiles to build one row of a wedge, but just one tile to propagate a message up one row, closer to the frontier. So, almost surely, the message will have unboundedly many opportunities to incorporate its information into the "inbuffer row"; hence, almost surely, the message will be delivered.   □
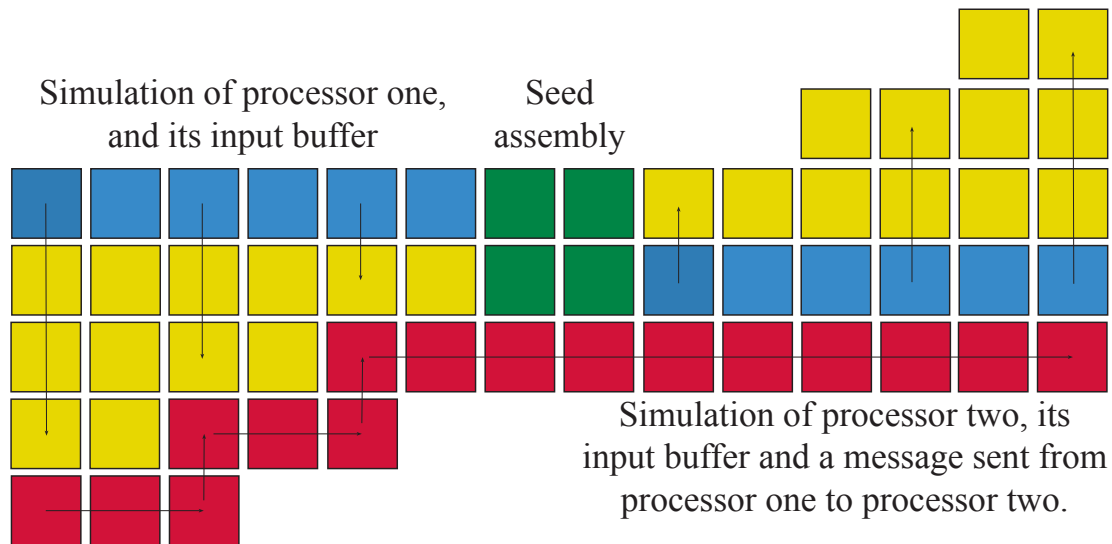
Figure 6: Schematic of how to simulate a two-processor message-passing system with two-dimensional tiling. The green tiles are the initial seed assembly, out of which the yellow wedges grow. The wedges simulate the processors. An additional tile is attached to each level of the wedge, to simulate the processor's inbuffer, as described further in Figure 7. To simulate sending a message from processor 1 to processor 2, the wedge on the left sends a message down the side that does not contain the inbuffer. The transmission of that message is done by the tiles colored in red. (The black arrows indicate the order in which the red tiles bind during assembly.) If processor 2 sent a message to processor 1, we could simulate this in tiles by constructing a similar "red" transmission from the northeast edge of the wedge simulating processor 2, westward toward processor 1 (essentially a mirror image of the red tiles in the schematic).
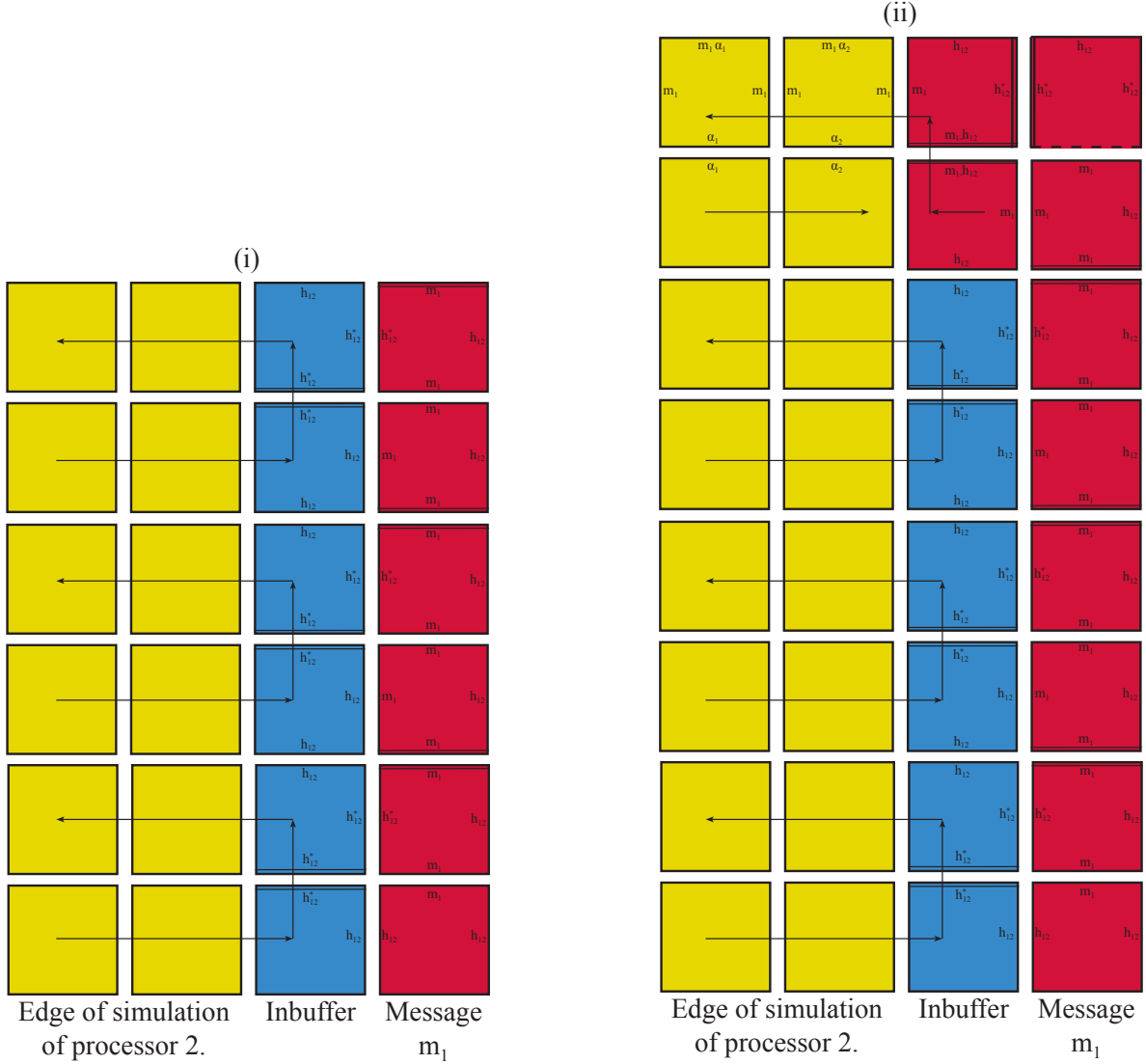
Figure 7: This figure illustrates the simulation of a processor, its inbuffer, and the receipt of a message. The yellow tiles are the edge of a "wedge" that simulates processor 2 (in the simulation of a two-processor system). The blue tiles represent the inbuffer of processor 2. In this diagram, every other row of the wedge is a row that is built from the inbuffer tile, to the west. The arrows show the direction that tiles bind in. More generally, the wedge construction does not need to "check" the inbuffer at every other row, as long as it checks infinitely often. Figure (i) shows how message m_1 attaches along the east side of the inbuffer, and has opportunities due to double bonds to bind to the north of the most recent inbuffer tile. In Figure (ii), m_1 is successfully transferred into the inbuffer, which then means that the next row that checks the contents of the inbuffer transmits m_1 to (potentially) all tiles in the wedge, as shown. The farthest northeast tile in Figure (ii) changes the column to an inbuffer column, in case later messages need to be transferred from the east to the wedge growing to the west.
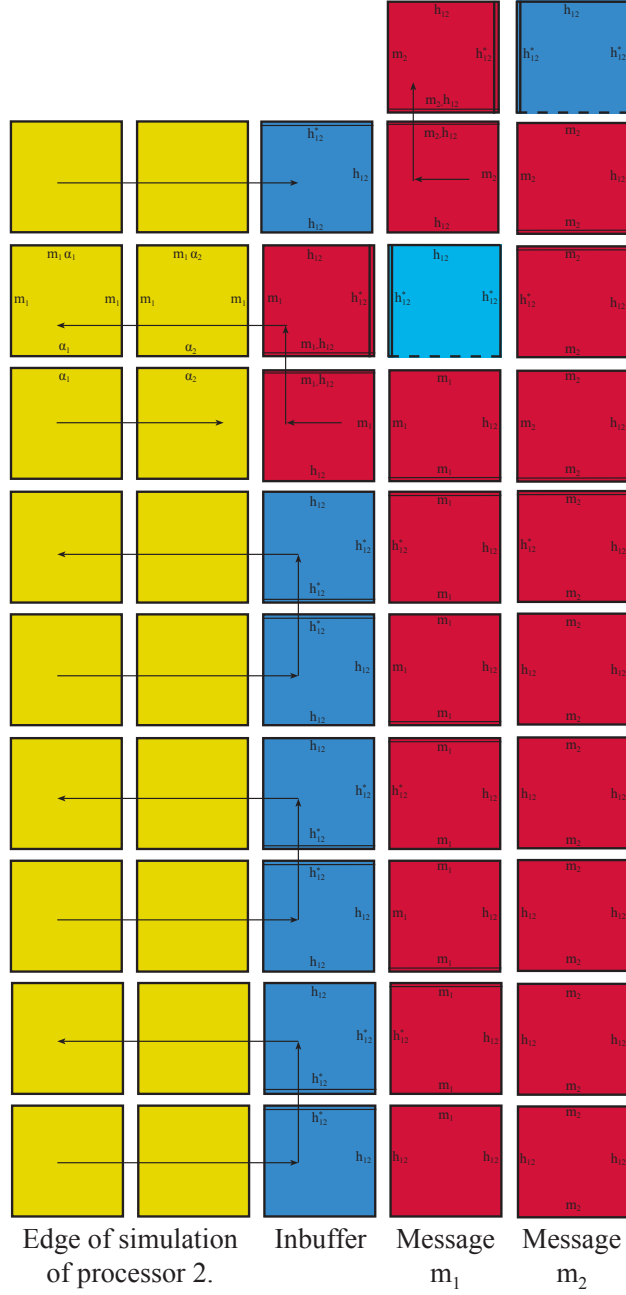
Figure 8: This figure continues the construction from Figure 7(ii), by showing how a second message, named m_2, gets transferred to the "first message" column of the wedge simulating processor 2. The mechanism shown in Figure 7 will then transfer m_2 to the inbuffer. Note that the furthest northeast tile sets up the configuration so any further messages sent can be transferred west, as m_2 was.